TracNav

- JPFWiki - Welcome Page
- **Introduction...**
- **Installing JPF...**
- **User Guide**
    - Application Types
    - JPF Components
    - Configuring JPF
    - Running JPF
    - JPF Output
    - The JPF API
- **Developer Guide...**
- **Projects...**
- Summer Projects
- External Projects
- Change(B)log
- **About...**
- Events
- Presentations
- Papers
- FAQ
- History?
- Support
- People?
- Playground
- Table of Context

Running JPF

There are five general ways to run JPF, depending on your execution environment (command prompt or IDE) and desired level of configuration support. This page has to cover quite some ground, so bear with us

1. from a command prompt (operating system shell)
2. from an IDE (!NetBeans, Eclipse) without using JPF plugins
3. from an IDE with JPF plugins installed
4. from within a JUnit test class
5. explicitly from an arbitrary Java program

1. Command Line

There are several ways to run JPF from the command line, using varying degrees of its runtime infrastructure. The most simple way is to use the provided `bin/jpf` script of the jpf-core distribution. Go to the directory where your SUT classes reside, and do a

```
> <jpf-core-dir>/bin/jpf +classpath=. <application-main-class>
```

or preferably

```
> <jpf-core-dir>/bin/jpf <application-property-file>.jpf
```

(see target specification below). If you want to avoid platform specific scripts, you only have to slightly expand this to

```
> java -jar <jpf-core-dir>/build/RunJPF.jar +classpath=. <application-main-class>
```

This makes use of the small *RunJPF.jar* startup jar that is part of the jpf-core distribution, which only includes the classes that are required to start the JPF bootstrapping process (esp. the JPF classloader). These classes automatically process the various JPF configuration files. If your SUT is not trivial, it is also recommended to add a "-Xmx1024m" host VM option, to avoid running out of memory.

Last (and probably most rarely), you can directly start JPF and give it an explicit classpath. This amounts to something like

```
> java -classpath <jpf-core-dir>/build/jpf.jar:<jpf-core-dir>/lib/bcel.jar gov.nasa.jpf.JPF \
                                          +classpath=. <application-main-class>
```

Of course, this gets quickly more complicated if you use JPF extensions, which require to add to both the host VM and the JPF classpath, which is completely automated if you use the RunJPF.jar method. Explicitly setting paths is only for rare occasions if you develop JPF components yourself.

There are three different argument groups that are processed by JPF:

(1) JPF command line options

These options should come first (after RunJPF.jar), and all start with a hyphen ("**-**"). The set of currently supported options is:

- "-help" : show usage information and exit
- "-log" : print the configuration steps
- "-show" : print the configuration dictionary after configuration is complete

The last two options are mostly used to debug if the JPF configuration does not work as expected. Usually you start with "-show", and if you don't see the values you expect, continue with "-log" to find out how the values got set.

(2) JPF properties

This is the second group of options, which all start with a plus ("**+**") marker, and consist of "+<key>=<value>" pairs like

```
.. +cg.enumerate_random=true
```

All properties from the various JPF properties configuration files can be overridden from the commandline, which means there is no limit regarding number and values of options. If you want to extend an existing value, you can use any of the following notations

- `+<key>+=<value>` - which appends <value>
- `++<key>=<value>` - which prepends <value>

- +<key>=..${<key>}.. - which gives explicit control over extension positions

Normal JPF properties "${<key>}" expansion is supported.

If the "=<value>" part is omitted, a default value of "true" is assumed. If you want to set a value to null (i.e. remove a key), just skip the <value> part, as in "+<key>="

(3) target specification

There are two ways to specify what application JPF should analyze

- explicit classname and arguments
  ```
  > jpf ...  x.y.MyApplication arg1 arg2 ..
  ```
- application property file (*.jpf)
  ```
  > jpf ... MyApplication.jpf
  ```

We recommend using the second way, since it enables you to store all required settings in a text file that can be kept together with the SUT sources, and also allows you to start JPF from within NetBeans or Eclipse just by selecting the *.jpf file (this is mainly what the IDE plugins are for). Please note that application property files require a "target" entry, as in

```
# JPF application property file to verify x.y.MyApplication
target = x.y.MyApplication
target_args = arg1,arg2
...
```

2. Running JPF from within IDE without plugins

You can start JPF from within NetBeans or Eclipse without having the IDE specific JPF plugins installed. In this case, JPF uses the standard IDE consoles to report verification results. For details, please refer to the following pages:

- Running JPF from within NetBeans without plugin
- Running JPF from Eclipse without plugin

Note that this is **not** the recommended way to run JPF from within an IDE, unless you want to debug JPF or your classes.

3. Running JPF from within IDE with plugins

You can simplify launching JPF from within NetBeans or Eclipse by using the respective plugins that are available from this server. In this case, you just have to create/select an application property (*.jpf) file within your test project, and use the IDE context menu to start a graphical JPF user interface. These so called "JPF shells" are separate applications (that can be configured through normal JPF properties), i.e. appear in a separate window, but can still communicate with the IDE, e.g. to position editor windows. You can find more details on

- Running JPF from within NetBeans with netbeans-jpf plugin
- Running JPF from Eclipse with eclipse-jpf plugin

This is becoming the primary method of running JPF. The benefits are twofold: (1) this is executed outside of the IDE process, i.e. it doesn't crash the IDE if JPF runs out of memory, and (2) it makes use of all your standard JPF configuration (site.properties and jpf.properties), in the same way like running JPF from a command line.

4. Launching JPF from JUnit tests

JPF comes with JUnit based testing infrastructure that is used for its own regression test suite. This mechanism can also be used to create your own test drivers that are executed by JUnit, e.g. through an Ant build script. The source structure of your tests is quite simple

```
import gov.nasa.jpf.util.test.JPFTestSuite;
import org.junit.Test;

public class MyTest extends JPFTestSuite {

  public static void main(String[] args) throws InvocationTargetException {
    runTestsOfThisClass(args);
  }

  @Test
  public void testSomeFunction() {
    if (verifyNoPropertyViolation()) {  // or a number of other JPFTestSuite defined goals
       someFuntction(); ..              // this section is verified by JPF
    }
  }

  //.. more @Test methods
```

From a JUnit perspective, this is a completely normal test class. You can therefore execute such a test with the standard `<junit>` Ant task, like

```
<property file="${user.home}/.jpf/site.properties"/>
    <property file="${jpf-core}/jpf.properties"/>
    ...
    <junit printsummary="on" showoutput="off" haltonfailure="yes"
           fork="yes" forkmode="perTest" maxmemory="1024m">
      ...
      <classpath>
        ...
        <pathelement location="${jpf-core}/build/jpf.jar"/>
      </classpath>

      <batchtest todir="build/tests">
        <fileset dir="build/tests">
          ...
          <include name="**/*Test.class"/>
        </fileset>
      </batchtest>
    </junit>
    ...
```

Only jpf.jar needs to be in the host VM classpath when compiling and running the test, since `gov.nasa.jpf.util.test.JPFTestSuite` will use the normal JPF configuration (site.properties and configured jpf.properties) to set up the required `native_classpath`, `classpath` and `sourcepath` settings at runtime. Please refer to the JPF configuration page for details.

If you want explicit control over the host VM classpath (JPF's `native_classpath` setting), you can use `gov.nasa.util.test.TestJPF` as the base class for your test (which doesn't use the JPF classloader), but in this case you need to add all jars required by all JPF components you need for your test (i.e. `jpf-<project>/build/jpf-<project>.jar` and `jpf-<project>/lib/*.jar` for all required JPF projects).

If you don't have control over the build.xml because of the IDE specific project type (e.g. if your SUT is configured as a NetBeans "Class Library Project"), you have to add jpf.jar as an external jar to your IDE project configuration.

In addition to adding jpf.jar to your build.xml or your IDE project configuration, you might want to add a jpf.properties file to the root directory of your project, to set up things like where JPF finds classes and sources it should analyze (i.e. settings that should be common for all your tests). A generic example could be

```
# example of JPF project properties file to set project specific paths

  # no native classpath required if this is not a JPF project itself

  # where does JPF find the classfiles to execute
  classpath=build/classes;build/test/classes

  # where are the sources, in case JPF wants to create a trace
  sourcepath=src;test

  # other project common JPF settings like autoloaders etc.
  listener.autoload+=,javax.annotation.Nonnull
  listener.javax.annotation.Nonnull=.aprop.listener.NonnullChecker
  ...
```

You can find project examples here

- standard NetBeans project ("Java Class Library" or "Java Application")
- Freeform NetBeans project? (with user supplied build.xml)
- standard Eclipse project? (with user supplied build.xml)

Please refer to the Verify API and the JPF tests pages for details about JPF APIs (like `verifyNoPropertyViolation(..)` or `Verify.getInt(min,max)`) you can use within your test classes.

Since JPF projects use the same infrastructure for their regression tests, you can find a wealth of examples under the `src/tests` directories of your installed JPF projects.

5. Explicitly Launching JPF from a Java Program

Since JPF is a pure Java application, you can also run it from your own application. The corresponding pattern looks like this:

```
public class MyJPFLauncher {
  ...
  public static void main(String[] args){
    ..
    try {

      // this initializes the JPF configuration from default.properties, site.properties
      // configured extensions (jpf.properties), current directory (jpf.properies) and
      // command line args ("+<key>=<value>" options and *.jpf)
      Config conf = JPF.createConfig(args);

      // ... modify config according to your needs
      conf.setProperty("my.property", "whatever");

      // ... explicitly create listeners (could be reused over multiple JPF runs)
      MyListener myListener = ...

      JPF jpf = new JPF(conf);

      // ... set your listeners
      jpf.addListener(myListener);

      jpf.run();
      if (jpf.foundErrors()){
        // ... process property violations discovered by JPF
      }
    } catch (JPFConfigException cx){
      // ... handle configuration exception
      // ...  can happen before running JPF and indicates inconsistent configuration data
    } catch (JPFException jx){
      // ... handle exception while executing JPF, can be further differentiated into
      // ...  JPFListenerException - occurred from within configured listener
      // ...  JPFNativePeerException - occurred from within MJI method/native peer
      // ...  all others indicate JPF internal errors
    }
  ...
```

Please refer to the Embedding JPF developers documentation for details. If you start JPF through your own launcher application, you have to take care of setting up the required `CLASSPATH` entries so that it finds your (and JPFs) classes, or you can use the generic `gov.nasa.jpf.Main` to load and start your launcher class, which makes use of all the path settings you have in your site.properties and the directories holding project properties (`jpf.properties`) referenced therein (details on how to configure JPF. This brings us back to the command line at the top of this page, only that you specify which class should be loaded through `Main`:

```
> java -jar .../RunJPF.jar -a MyJPFLauncher ...
```

(note that `gov.nasa.jpf.Main` is the `Main-Class` entry of the executable `RunJPF.jar`, which also holds the `JPFClassLoader`).

Just for the sake of completeness, there is another way to start JPF explicitly through a gov.nasa.jpf.JPFShell? implementation, which is using the normal `JPF.main()` to load your shell, which in turn instantiates and runs a `JPF` object. This is specified in your application property (*.jpf) file with the `shell=<your-shell-class>` option. Use this if your way to start JPF is optional, i.e. JPF could also be run normally with your *.jpf. The graphical JPF shell is an example for this.